Rapport Projet Données Réparties

Un service de partage d'objets répartis et dupliqués en Javas

GRETHEN Clementine
WEISBECKER Lisa

Table des matières

Introduction

Le but de ce projet est d'illustrer les principes de programmation répartie vus en cours. Pour ce faire, nous allons réaliser sur Java un service de partage d'objets par duplication, reposant sur la cohérence à l'entrée (entry consistency). Les applications Java utilisant ce service peuvent accéder à des objets répartis et partagés de manière efficace puisque ces accès sont en majorité locaux (ils s'effectuent sur les copies (réplicas) locales des objets). Durant l'exécution, le service est mis en œuvre par un ensemble d'objets Java répartis qui communiquent au moyen de Java/RMI pour implanter le protocole de gestion de la cohérence.

1 Présentation du service

Le schéma fournit ci-dessous présente l'architecture du service utilisé pour le projet.

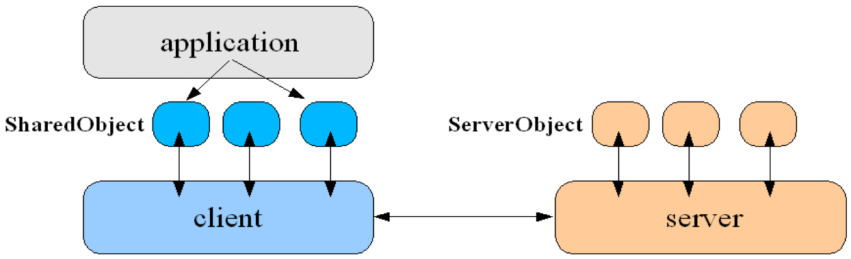

FIGURE 1 – Architecture du service

Il s'agit d'un service de gestion des objets partagés en Java qui permet de gérer la cohérence des données à travers des descripteurs d'objets (instances de la classe SharedObject) qui contiennent un champ "obj" qui pointe sur l'instance Java partagée. Les objets partagés sont accessibles uniquement via cette indirection.

La classe SharedObject fournit des méthodes comme lock_read(), lock_write() et unlock() pour gérer la cohérence des données. Le service utilise également un serveur de noms pour créer, retrouver et enregistrer des objets partagés.

La classe Client fournit des services pour créer ou retrouver des objets dans le serveur de noms, et permet d'initialiser la couche cliente au démarrage de l'application.

Ce projet est séparé en 3 étapes dinstinctes :
La première étape consiste à implanter le service de gestion d'objets partagés répartis. Nous avons défini un shéma de synchronisation pour l'accès à ces objets.

Dans la deuxième étape, on introduit les "stubs" pour soulager le travail du programmeur. Ces stubs auront une interface qui correspond à celle de l'objet partagé original. Les applications utiliseront ces stubs plutôt que les SharedObjects pour accéder aux objets partagés.

Pour la troisième étape, il a fallu prendre en compte le stockage de références à des objets partagés dans d'autres objets partagés. Pour cela, nous devons adapater les primitives de sérialisation des stubs pour que lorsqu'un stub est sérialisé sur une machine, il puisse être correctement désérialisé sur une autre machine et pointer vers le bon stub de l'objet partagé lié.

2 Réalisation des Etapes

2.1 Etape 1

2.1.1 SharedObject

La classe SharedObject implémente l'interface SharedObject_itf et permet de stocker un objet partagé avec un verrou pour gérer les accès concurrents à celui-ci. L'objet partagé est stocké dans l'attribut `obj` et l'identifiant de l'objet dans l'attribut `id`.

Le SharedObject peut prendre plusieurs verrous, un schéma simple permet de comprendre cette notion :

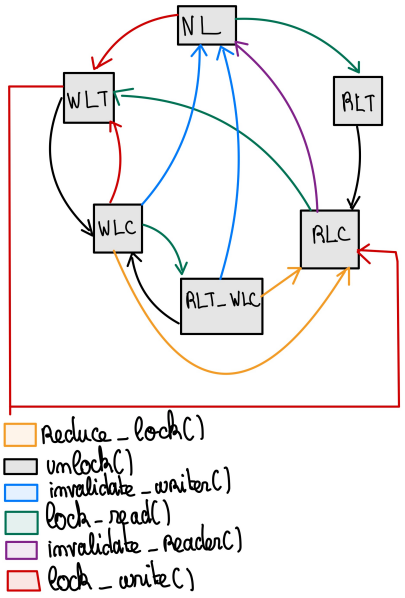

FIGURE 2 – Etats d'un SharedObject

L'état courant du verrou est stocké dans l'attribut `lock` qui est de type `Enum_lock`. Il peut prendre les valeurs suivantes : NL (No Lock), RLC (Read Lock Cached), WLC (Write Lock Cached), RLT (Read Lock Taken), WLT (Write Lock Taken) et RLT_WLC (Write Lock Taken - Write Lock Cached). L'attribut `attente` est utilisé pour bloquer le thread en cas d'accès concurrent.

La méthode `lock_write()` permet de verrouiller l'objet en écriture. Elle utilise le système de transitions entre les différents états du verrou pour gérer les différents cas d'utilisation. Si l'objet était dans l'état NL, elle envoie une demande de `lock_write` au client pour récupérer la version valide de l'objet.

La méthode `lock_read()` permet de verrouiller l'objet en lecture. Elle fonctionne de manière similaire à `lock_write()`, en gérant les différents cas d'utilisation et en envoyant une demande de `lock_read` au client si nécessaire pour récupérer la version valide de l'objet.

L'utilisation de "synchronized" dans le code permet de gérer les problèmes d'interblocage, qui peuvent se produire lorsque plusieurs threads tentent d'accéder simultanément à une ressource partagée. Lorsqu'une méthode est déclarée comme "synchronized", seul un thread à la fois peut y accéder. Si un autre thread tente d'accéder à cette méthode alors qu'elle est déjà en cours d'exécution par un autre thread, il sera bloqué jusqu'à ce que la méthode soit libérée. Cela empêche les threads d'entrer en conflit lors de l'accès à une ressource partagée, ce qui évite les problèmes d'interblocage. Ici, les méthodes `lock_write()` et `lock_read()` utilisent `synchronized` pour s'assurer qu'il n'y a qu'un seul thread qui peut accéder à la section critique de code à un moment donné. Le mot clé synchronized est utilisé pour mettre en œvre le concept de moniteur. Le but est de maintenir un accès privilégié sur une ressource critique. Comme en cours de système concurrent, nous avons également les méthodes wait() et notify().

La méthode `unlock()` permet de déverrouiller l'objet en mettant à jour son état et en notifiant les autres threads qui attendent sur cet objet.

Les méthode `invalidate_writer()` et `invalidate_reader()` ont pour but de rendre l'objet partagé inaccessible aux autres clients qui ont pris un verrou en écriture/lecture sur celui-ci.

En résumé, cette classe a permit de mettre un oeuvre le shéma de synchronisation sur les objets partagés et de gérer les accès.

2.1.2 ServerObject

La classe ServerObject est utilisée pour gérer les accès concurrents à un objet partagé sur le serveur. Elle implémente l'interface ServerObject_itf, qui définit les méthodes nécessaires pour gérer les verrous en lecture et en écriture sur l'objet partagé.

Les principaux attributs de cette classe sont :

- lecteurs : une liste des clients ayant un verrou en lecture sur l'objet ;
- ecrivain : le client ayant un verrou en écriture sur l'objet ;
- obj : l'objet partagé ;
- id : l'identifiant de l'objet partagé ;
- lock : l'état courant de l'objet partagé (NL : No lock, WL : Write lock, RL : Read lock) ;

La méthode lock_read(Client_itf client) permet à un client de verrouiller l'objet en lecture. Elle utilise le système de transitions entre les différents états de l'objet (NL, WL, RL) pour gérer les différents cas d'utilisation. Si l'objet était verrouillé en écriture, elle va d'abord demander au client ayant le verrou en écriture de réduire son verrou avant de donner un verrou en lecture au client qui a fait la demande. Elle utilise une section synchronized pour s'assurer qu'il n'y a qu'un seul thread qui peut accéder à la section critique de code à un moment donné.

La méthode lock_write(Client_itf client) permet à un client de verrouiller l'objet en écriture. Elle gère les différents cas d'utilisation de la même manière que lock_read, en invalidant les verrous en lecture sur l'objet et en donnant un verrou en écriture au client qui a fait la demande. Elle utilise également une section synchronized pour s'assurer qu'il n'y a qu'un seul thread qui peut accéder à la section critique de code à un moment donné.

La méthode getId() retourne l'identifiant de l'objet.

En résumé, cette classe permet de gérer les accès concurrents à un objet partagé sur le serveur en utilisant des verrous en lecture et en écriture, en utilisant un système de transitions entre les différents états de l'objet, et en utilisant une section synchronized pour éviter les problèmes d'interblocage.

2.1.3 Server

La classe server est un serveur RMI qui permet de gérer des objets partagés. Elle implémente l'interface Server_itf qui définit les méthodes pour créer, verrouiller en lecture et verrouiller en écriture les objets partagés.

Les principaux attributs de cette classe sont :

— serverObjects : une HashMap qui contient les objets partagés, associant à chaque objet un identifiant unique ;

— registre : une HashMap qui permet de stocker les noms des objets partagés et leurs identifiants correspondants pour que les clients puissent les retrouver facilement ;

— id_en_cours : un entier qui contient l'identifiant actuellement utilisé pour identifier les objets partagés ;

La méthode create (Object obj) permet de créer un objet partagé. Elle crée un objet de la classe ServerObject, lui affecte un identifiant unique, et l'ajoute à la HashMap serverObjects.

Les méthodes lock_write (int id,Client_itf client) et lock_read (int id,Client_itf client) permettent respectivement de verrouiller un objet partagé en écriture et en lecture pour un client donné. Les méthodes vérifient d'abord que l'objet partagé existe bien en utilisant l'identifiant donné, puis utilisent les méthodes lock_write et lock_read de la classe ServerObject pour effectuer l'opération de verrouillage.

Il y a plusieurs choses qui peuvent être notées :

1. La classe utilise une HashMap pour stocker les objets partagés, ce qui permet une recherche rapide des objets en utilisant leur identifiant unique.
2. La classe server utilise des verrous pour gérer les accès concurrents aux objets partagés, ce qui permet de s'assurer qu'un seul thread peut accéder à un objet partagé à un moment donné.
3. L'utilisation de RMI permet de gérer les objets partagés à distance, c'est-à-dire que les clients peuvent accéder aux objets partagés depuis des ordinateurs différents.

La méthode main permet de lancer le server RMI. Cette méthode effectue les étapes suivantes :

— Définition de l'adresse IP du serveur (localhost)

— Création d'un objet serveur

— Enregistrement de l'objet serveur auprès du registre RMI (en utilisant Naming.rebind)

— Affichage d'un message pour indiquer que le serveur est prêt à recevoir des requêtes.

2.1.4 Client

La classe Client implémente l'interface Client_itf et étend UnicastRemoteObject. Elle contient plusieurs attributs statiques :

— instance : un objet de type Client_itf qui est l'instance unique de la classe. Elle est créée par la méthode init().
— server : un objet de type Server_itf qui est l'instance du serveur récupérée par la méthode lookup() de la classe Naming.
— objets_client est une HashMap qui stocke les objets partagés localement, qui ont été récupérés par le client.

La classe contient une méthode statique init() qui permet d'initialiser l'instance unique de la classe. Elle récupère également l'instance du serveur et initialise la HashMap.

lookup(String nom) récupère l'objet partagé de nom "nom" sur le serveur, l'ajoute à la HashMap des objets partagés locaux et le retourne. Si l'objet n'existe pas, la méthode retourne null.

register(String name, SharedObject_itf objet_enregistre) enregistre l'objet partagé "objet_enregistre" de nom "name" sur le serveur.

create(Object objet_creation) prend en paramètre un objet "objet_creation" qui sera partagé. Elle utilise la méthode create() de la classe Server pour créer un objet ServerObject associé à un id sur le serveur. Cet id est ensuite utilisé pour créer un objet SharedObject qui est ajouté à la HashMap des objets partagés du client. Cela permet de stocker localement les objets partagés pour pouvoir les utiliser plus facilement par la suite.

Il y a aussi des méthodes lock_write(int id) et lock_read(int id) pour réaliser les demandes de verrou en écriture ou en lecture sur un objet partagé.

2.1.5 Tests

Test IRC

Pour nos tests, nous avons testé de trois manières différentes afin de prouver trois niveaux de réalisation. Dans un premier temps, nous nous sommes servis du fichier Irc.java pour s'assurer que l'écriture et la lecture fonctionnent. Ce test ne montrera aucune situation d'interblocage, car on ne peut pas avoir un grand débit d'information. Ainsi voilà les résultats : Au niveau le plus bas, un lecteur

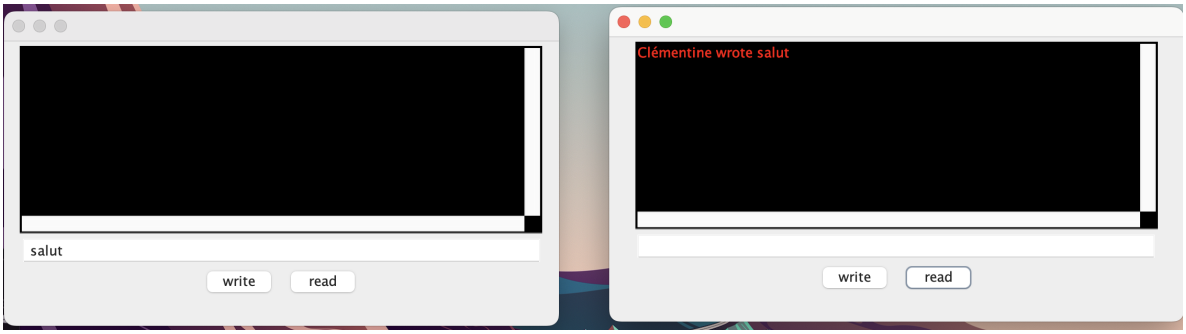

FIGURE 3 – Test basique entre un écrivain et un lecteur

arriver à lire un écrivain sans soucis.

Tests avec Sentence

Au niveau le plus bas, un lecteur arriver à lire un écrivain sans soucis. Pour vraiment montrer la cohérence de notre synchronisation, nous avons implanté deux fichiers : testBoucleReader.java et TestBoucleWriter.java. Un lecteur va lire en continue et un écrivain va lire en continue (affichage dans le terminal : "start" pour commencer à lire "s" pour commencer à écrire). Voilà nos résultats pour deux écrivains et un lecteur :

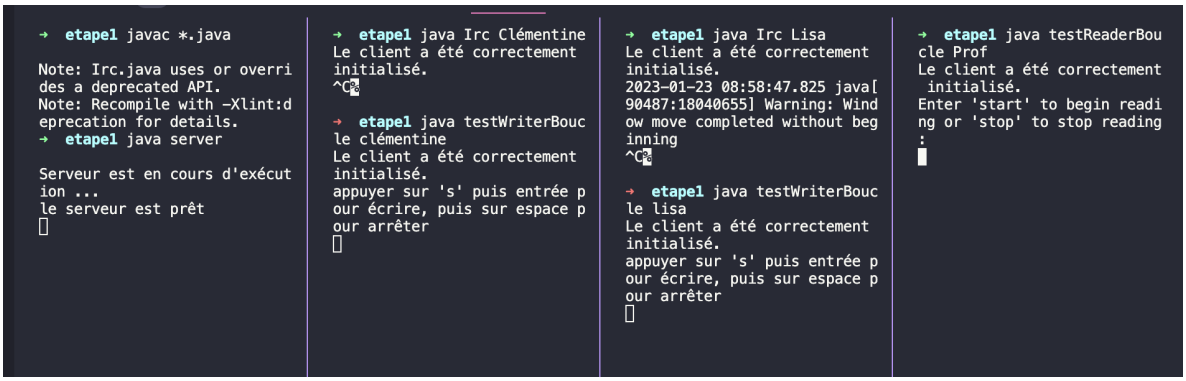

FIGURE 4 – Commande pour deux écrivains et un lecteur

Et nous obtenons :

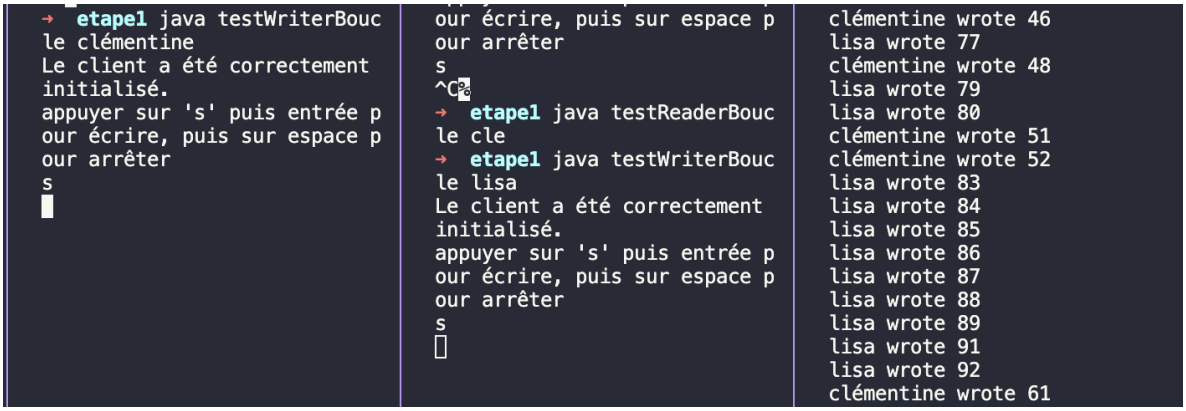

FIGURE 5 – Affichage

Nous remarquons qu'il n'y a pas d'interblocage.

Tests avec SentenceNumérique

Pour aller plus loin, nous avons voulu travailler sur un deuxième objet partagé de type SentenceNumérique (on travaille maintenant sur des Integer et les "écrivains" réalisent des opérations sur ces nombres.).
La classe SentenceNumerique, tout comme Sentence, représente un objet sérialisable (nécéssaire pour le partager). Elle implémente donc la classe "Serializable".

On a implémenté deux classes de tests (sur le même principe que Sentence) : testNombreReader.java et testNombreWriter.java. Nous obtenons les mêmes résultats que précédemment. Le but était donc de tester le fonctionnement pour d'autres type de SharedObject, mais également de les manipuler (cela a faciliter notre compréhension.).

2.2 Etape 2

2.2.1 Analyse de cette étape

L'étape précédente était basée sur la création de l'architecture complète du problème avec une utilisation explicite des SharedObject par les applications. Ici, on facilite le processus en réalisant un des stubs que les applications n'aient plus à manipuler de SharedObjects directement. Le fonctionnement général est : on se munit d'une interface Sentence itf, que n'implémente pas Sentence, qui dispose des méthodes de Sentence et hérite, en plus, de SharedObject itf et donc de ses méthodes de verrouillages. A partir de l'étape 2, l'utilisateur pourra se contenter de manipuler des Sentence itf et n'aura plus à passer par les SharedObects. Le but du générateur de stubs sera de fournir une implémentation de Sentence itf : Sentence stub, qui héritera en plus de SharedObject. Nous ne revenons que sur les classes qui ont été modifiées.

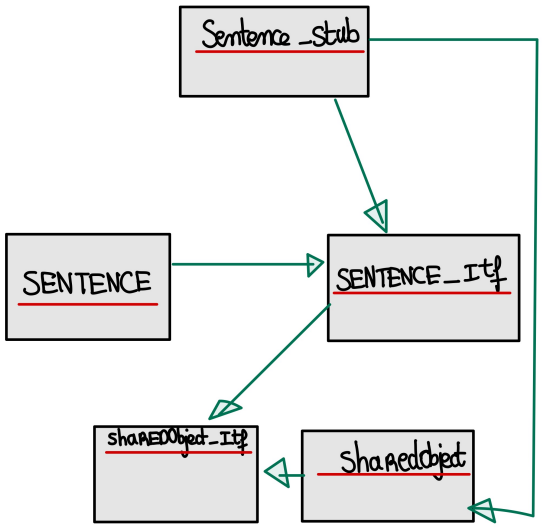

FIGURE 6 – Fonctionnement des stub

2.2.2 La classe StubGenerator

Cette classe est une nouvelle classe qui va permettre d'implanter le service de stub. On utilise la réflexion. Le but est de créer un fichier .java pour un stub qui étend la classe SharedObject et implémente l'interface objet donné. On regarde notamment, si le stub existe déjà et sinon on va le créer. Nous utilisons ici des methodes vues en métaprogrammation. A la fin, nous compilons le fichier généré. Cependant, sur une de nos machine de test, la compilation de fonctionne pas et genère un NullPointerException. Après plusieurs recherche, cela est causé par la version de Java. Donc si une erreur est relevée, il faut, après avoir exécuté StubGenerator (qui aura créé le fichier stub), le compiler à la main. Cela est contraignant, mais nous avons réalisé ce problème trop tardivement pour pouvoir penser à une autre manière de la coder (nous avons simplement fait une gestion d'exception.). Grâce à StubGenerator, nous allons pouvoir faire des appels sistants sans avoir à se soucier de la communication réseau.

La classe génère un fichier Java qui contient une classe qui hérite de la classe SharedObject et qui implémente l'interface de l'objet passé en paramètre. Elle récupère les méthodes de cette interface en utilisant la réflexion et écrit leur déclaration dans le fichier, avec les paramètres et les exceptions. Elle génère également un constructeur qui prend en paramètre l'objet et son id pour pouvoir utiliser les méthodes de la classe SharedObject

2.2.3 La classe Client

La classe Client utilise la réflexion pour générer des stubs à partir d'objets distants. Le but des stubs est de permettre aux clients d'appeler des méthodes sur ces objets distants.

La méthode lookup est utilisée pour récupérer un objet partagé distant en utilisant son nom. Si l'objet existe sur le serveur, elle va alors générer un stub pour cet objet en utilisant la méthode generateStub de la classe StubGenerator, puis elle va utiliser la réflexion pour construire une instance de l'objet partagé en utilisant le stub généré. Cette instance est ensuite ajoutée à une hashmap d'objets client pour être utilisée par la suite.

La méthode create de la classe Client permet de créer un objet partagé sur le serveur. Elle prend en paramètre le nom de l'objet partagé et la classe de l'objet. Elle utilise la réflexion pour créer un nouvel objet de la classe spécifiée et l'envoie au serveur pour qu'il soit créé en tant qu'objet partagé. Elle génère également un stub pour cet objet partagé en utilisant la classe StubGenerator et l'enregistre dans une HashMap de l'objet client.

Il n'y a pas de modification pour les autres classes.

2.3 Test

Nous n'avons pas réalisé de tests en plus, nous avons pris les mêmes test que pour l'étape 1 car le but est de vérifier que ça fonctionne de la même manière avec des stubs. Nous avons juste adapté nos classes de tests en remplaçant les types (ex SharedObjectItf par sentenceItf). Il faut prendre en compte le problème de compilation selon votre version de Java.

2.4 Etape 3

Après analyse de cette étape, nous avons rencontré des difficultés de réalisation et de compréhension. Nous avons donc fait le choix de ne pas implanter cette étape dans notre code mais de donner nos idées sur le rapport. En effet, nous avons fait le choix de nous concentrer sur l'optimisation de nos tests de la partie 1 et 2 et d'avoir un code bien réalisé et bien commenté, plutôt que d'avoir trois parties bancales.

2.4.1 Analyse

L'objectif de cette étape est de permettre aux objets partagés (SharedObject) de faire référence à d'autres objets partagés ou "stubs". Le problème est qu'il faut s'assurer que ces références pointent toujours sur la dernière version de l'objet partagé en question. Pour y arriver, il faut intervenir lors de la désérialisation de l'objet . Ensuite, il faut vérifier si le client possède déjà ces objets en utilisant leur identifiant. Si c'est le cas, il suffit de remplacer la référence par celle que le client connaît déjà. Sinon, il faut avertir le client qu'il doit créer un stub pour cet objet partagé. Nous pensons qu'il ne faut pas changer la synchronisation générale du projet. L'objectif est de réimplenter la méthode ReadResolve().

Ceci semble logique, car c'est elle qui est appelée à la désérialisation. D'après nos recherches, lorsqu'un objet est désérialisé, Java crée une nouvelle instance de l'objet en utilisant le constructeur par défaut. Cependant, si la classe contient une méthode readResolve() définie, cette méthode est appelée immédiatement après la création de l'objet pour permettre à la classe de fournir une instance différente de celle créée par défaut. C'est sur ce dernier aspect qu'il faut travailler : rédéfinir ReadResolve() afin de réaliser cette étape. Nous pensons qu'il faut la coder dans SharedObject. Cependant, nous avons pensé qu'il y'a davantage de choses à réaliser, en effet, la cohérence des Stubs n'intervient que pour un Client (le Server ne s'en occupe pas). Donc il est nécessaire de savoir si la Désérialisation de l'objet est réalisée dans le Client ou dans le Server (il faudrait créer des methodes annexes pour cela.).

Par manque de temps et de méthode de réalisation plus précise, nous avons donc décidé de ne pas poursuivre.

3 Conclusion

Ce projet était intéressant pour comprendre, et surtout mettre en lien les objets d'intergiciel et de système concurrent. Cependant, nous pensions plusieurs fois avoir tout compris et des éléments ont à chaque fois remis en cause cela, ce qui nous a pris beaucoup de temps. Au final, nous avons réussi à implanter la base du projet, et des tests intéressants. Même si, nous pensons qu'ils ne couvrent pas la totalité des critères, car nous n'avons pas forcément compris l'intégralité des choses à tester. Nous avons compris globalement l'enjeu de la partie 3 et réalisé une réflexion pour une réponse, mais nous n'avons pas continué. Globalement, ce projet fut très utile pour comprendre les nouveaux objets de Java. Pour l'organisation, nous avons travaillé sur Github et utilisé VSCode Live SHare.